CH7 Discussion Review

Eng. Tamer N. Jarada

Q1

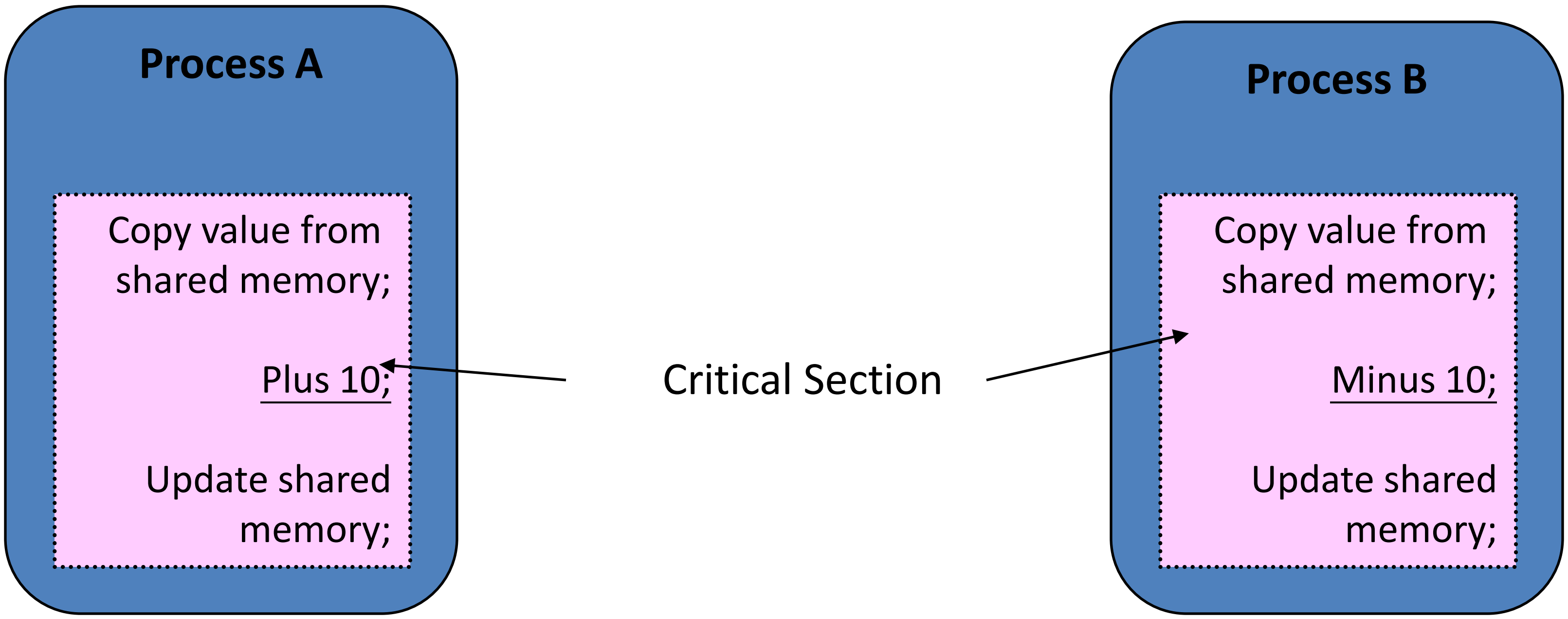
- What is a Race Condition? How could we prevent that?
  - Race condition is the situation where several processes or threads access and manipulate shared data concurrently. The final value or the result depends upon the order of processes' execution, i.e., which process finishes last.
  - To prevent race conditions, concurrent processes must be synchronized or atomic operations need to be ensured.

Q2

- What is Atomic operation? Why is that important?
  - An operation that cannot be interrupted during its execution. Without this, it is nearly impossible to maintain consistent state for shared variables involved more than one process or thread.

Critical Section

- Different processes have their own critical sections.
  - The tasks inside the critical sections are not the same.

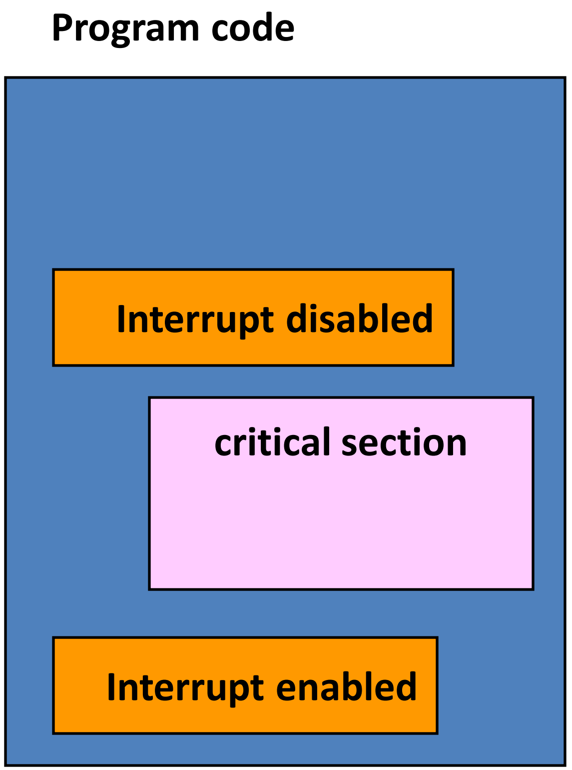


Q3.

- What are the three conditions that a solution to the critical section problem must guarantee?
  - Mutual exclusion
    - if a process is accessing a shared object, other processes must be excluded from accessing the same shared object
  - Progress
    - If no process is in its CS and one or more processes that wish to enter their CS, it must be possible for those processes (not in its remainder section) to negotiate who will proceed next into CS
  - bounded waiting
    - After a process has made a request to enter CS OTHER processes have a limited number of times that they can enter CS ((no process need to wait forever))

Ways to do Mutual Exclusion

- Hardware solution
  - Disabling interrupts
    - → disable context switching inside the critical section.
  - Correct but not attractive solution:
    - Terrible to let user programs to disable/enable interrupts.
    - Not working in multiprocessor systems.
    - May disable system clock



Ways to do Mutual Exclusion

- Software solution
  - Peterson’s solution
    - as an example to illustrate the idea
    - restricted to two processes
  - Bakery algorithm
    - For multiple processes

  - Semaphores
  - Monitors

Peterson's solution

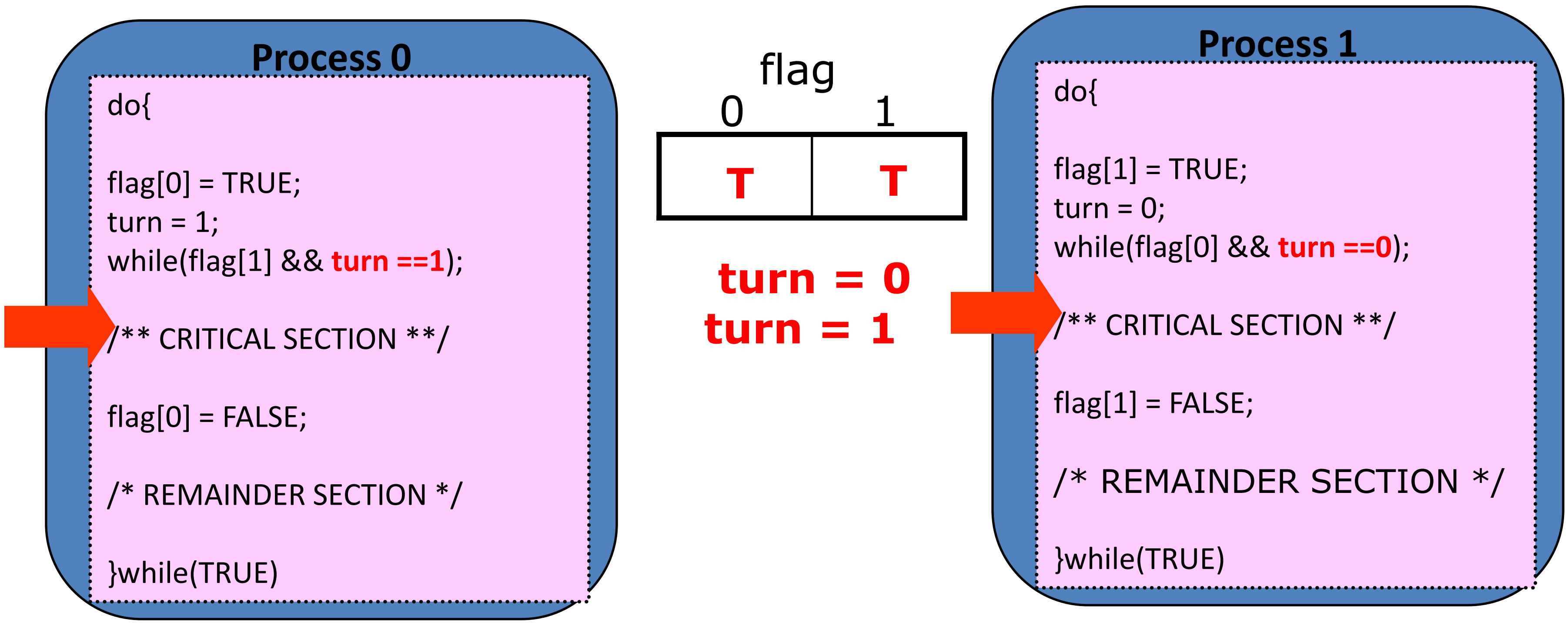
- Two processes solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.

```
do {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);
            CRITICAL SECTION
        flag[i] = FALSE;
              REMAINDER SECTION
    } while (TRUE);
```

- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Proof of Peterson’s solution

- Mutual exclusion is preserved
  - If both processes are in critical section
    - Then *flag*[0] = *flag*[1] = TRUE
      - → satisfying the first condition of the *while* loop
    - But *turn* can only be 0 or 1
      - → one of the processes must have to wait.
  - So impossible to have both processes in critical section

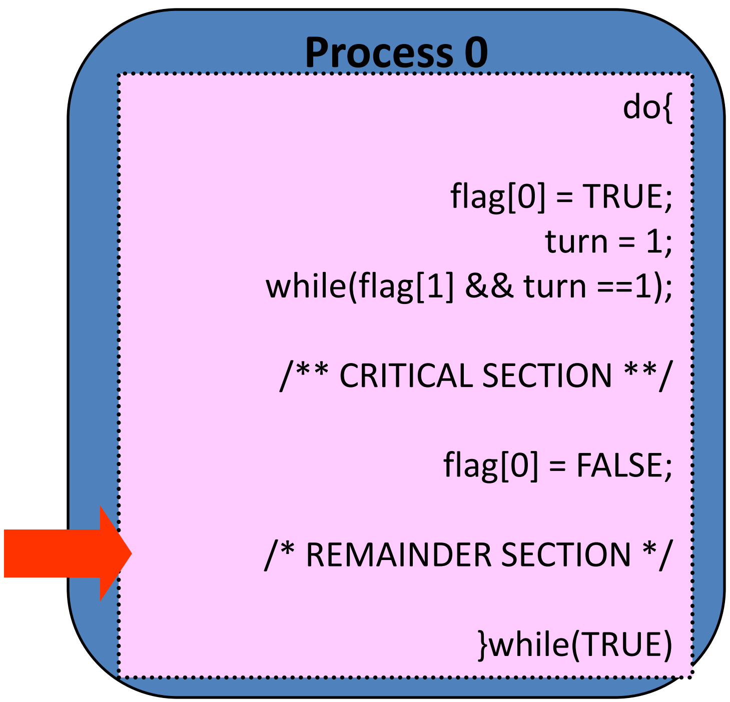


Proof of Peterson's solution

- Progress requirement
  - **If no process is in its CS and**

    one or more processes that wish to enter their CS, it must be possible for those processes (not in its remainder section)

    to negotiate who will proceed next into CS



- Use *turn* to decide which process will enter CS
- If a process is in RS, can't set *turn*, that is can't take part in the decision

Proof of Peterson's solution

- Bounded-waiting requirement
  - After a process has made a request to enter CS
  - OTHER processes have a limited number of times (1 time in Peterson's solution) that they can enter CS
  - → no process need to wait forever

Process 0

```
do{

flag[0] = TRUE;
turn = 1;
while(flag[1] && turn ==1);

/** CRITICAL SECTION **/

flag[0] = FALSE;

/* REMAINDER SECTION */

}while(TRUE)
```

Process 1

```
do{

flag[1] = TRUE;
turn = 0;
while(flag[0] && turn ==0);

/** CRITICAL SECTION **/

flag[1] = FALSE;

/* REMAINDER SECTION */

}while(TRUE)
```

Proof of Peterson's solution

- Bounded-waiting requirement
  - After a process has made a request to enter CS
  - OTHER processes have a limited number of times (1 time in Peterson's solution) that they can enter CS
  - → no process need to wait forever

Process 0

```
do{

flag[0] = TRUE;
turn = 1;
while(flag[1] && turn ==1);

/** CRITICAL SECTION **/

flag[0] = FALSE;

/* REMAINDER SECTION */

}while(TRUE)
```

Process 1

```
do{

flag[1] = TRUE;
turn = 0;
while(flag[0] && turn ==0);

/** CRITICAL SECTION **/

flag[1] = FALSE;

/* REMAINDER SECTION */

}while(TRUE)
```

Question 7.4 (Decker Algorithm)

- P0 and P1, share the following variables:
  - boolean flag[2]; /* initially false */
  - int turn;
- The structure of process Pi (i == 0 or 1) and Pj (j == 0 or 1) is the following:

```
while (true) {
    flag[i] = true;
    while (flag[j]) {
        flag[i] = false;
        while(turn == j);
        flag[i] = true;
    }
    critical();
    turn = j;
    flag[i] = false;
    remainder();
}
```

Prove that the algorithm satisfies all three requirements for the critical-section problem.

Ans 7.4 Mutual exclusion

- Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed. Namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.

Ans 7.4 Progress

- Progress is provided, again through the flag and turn variables. if a process wishes to access its critical section, it can set its flag variable to true and enter their critical section. It only sets turn to the value of the other process upon exiting its critical section.

Ans :7.4 Bounded waiting

- Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true, however only the thread whose turn it is can proceed, the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered - and exited - its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

Question 7.5 (Eisenberg McGuire Algorithm)

Q5

- What is the meaning of the term busy waiting?
  - a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor.
  - Alternatively, a process could wait by relinquishing the processor, and block on a condition (e.g., I/O, semaphore) and wait to be awakened at some appropriate time in the future.

Q6

- Can busy waiting be avoided altogether? Explain your answer?.
  - Busy waiting can be avoided ***but*** increase the overhead
  - putting a process to sleep and having to wake it up when the appropriate program state is reached.

Q7. Semaphore

- The signal() operation is used with semaphores and monitors. Explain the key difference in its runtime behavior in the two cases. (Hint: consider how this affects the wait() operation in another process) ?
  - In semaphores, every signal results in a corresponding increment of the semaphore value even if there are no process waiting. A future wait() operation could immediately succeed because of the earlier increment
  - When the signal() operation is used in monitors, if a signal is performed and if there are no waiting processes, the signal is simply ignored and the system does not remember the fact that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks.

Q8

```
semaphore mutex, empty, full;
mutex=1;
empty=0;            //Should be empty=N
full=N;             //Should be full=0
```

```
Producer:
do{
        ...
        // Produce an item in nextp
        ...
    wait(mutex);          // Order should
wait(empty);              // be switched
        ...
    // Add nextp to buffer
        ....
        signal (mutex);
    signal (full);
}while(true);
```

```
Consumer:
do{
        wait (mutex);     // Order should
    wait (full);          // be switched
        ...
        // remove an item from
    buffer to nextc
        ...
        signal (mutex);
    signal (empty);
        ...
        // consume the item in nextc
        ...
} while(true);
```

Q9

Writer	Reader
do {	do {
wait (wrt) ;	wait (mutex) ;
// writing is performed	readcount ++ ;
signal (wrt) ;	if (readcount == 1) wait(wrt) ;
} while (true)	signal (mutex)
	// reading is performed
	wait (mutex) ;
	readcount - - ;
	if (readcount == 0) signal(wrt) ;
	signal (mutex) ;
	} while (true)

1)What is the purpose of the semaphore "wrt"?

To guarantee mutual exclusion to the critical section

1)What is the purpose of the semaphore "mutex"?

To guarantee mutual exclusion when updating the shared variable readcount

1)Suppose a writer process is inside its critical section, while another writer and n readers are waiting outside their critical sections. Which semaphores are they waiting on, respectively?

the writer is waiting on wrt , the 1st reader is waiting on wrt and the other n-1 readers are waiting on mutex